```lua
-- For simple XAI (explainable AI), try a little sampling theory and a
-- little learning.
--
-- For example, if we apply a sorting heuristic to data, we can binary
-- chop our way down to good solutions. Assuming such chops,
--  at probability _P_, we find _q_
-- percent "best" items (where "best" is
-- defined by the Zitzler's multi-objective indicator) using
-- `n=log2(log(1-P)/log(1-q))` samples. e.g. the 5% best within 10,000 samples
-- is hunted down using less than n=10 samples.  Sounds too good to be true?
-- Well lets check.
--
-- This code starts with a config variable (`the`)
-- and ends with a library of demos (see the `go` functions at end of file).
-- Each setting can be (optionally) updated by a command-line flag.
-- Demos can be run separately or  all at once (using `-g all`).
-- For regression tests, we report the failures seen when the demos run.
--
-- <img src="xai4.jpeg" width=200 align=left>
--
-- This code makes extensive use of a DATA object.  Data from disk
-- becomes a DATA. DATA  are recursive bi-clustered by partitioning on
-- the distance to two distant ROWs (found via the FASTMAP
-- linear time random
-- projection algorithm).  Each cluster is new DATA object, containing a subset
-- of the data. A decision tree is built that reports the difference
-- between the "best" and "worst" clusters (defined using a multi-objective
-- domination predicate) and that tree is just a  tree
-- of DATAs with `kids` pointer to sub-DATAs).  This process
-- only needs log2(N) queries to y-values (while clustering,
-- just on the pairs of
-- distance objects).
--
-- convenntions "is" [refix is a bookeam. "n" is a number, sprefix=string
-- _ prefix means internal function
local _=require"lib" -- must be first line
local help=[[

XAI: Multi-objective semi-supervised explanation
(c) 2022 Tim Menzies <timm@ieee.org> BSD-2 license

USAGE: lua xiago [OPTIONS]

OPTIONS:
 -B  --Balance  for delta, ration rest:best               = 4
 -b  --bins     for bins, initial #bins (before merging) = 16
 -F  --Far      for far, how far to look for distant pol = .95
 -f  --file     data csv source           = ../data/auto93.csv
 -g  --go       start-up action                          = pass
 -h  --help     show help                                = false
 -m  --min      for half, cluster down to n^min          = .5
 -r  --ratios   for RATIO, max sample size               = 512
 -p  --p        for dist, distance coefficient           = 2
 -s  --seed     random number seed                       = 10019
 -S  --Some     for far, how many rows to explore        = 512
 -s  --stop     for delta, min row size                  = 6

Boolean flags need no arguments e.g. "-h" sets "help" to "true".]]

---- ---- ---- ---- Names
local the={}
help:gsub("\n [-][%S]+[%s]+[-][-]([%S]+)%s[^\n]+= ([%S]+)",
          function(k,x) the[k] = _.coerce(x) end)

---- Misc general functions
local any,big,cat,chat,cli,coerce    = _.any,_.big,_.cat,_.chat,_.cli,_.coerce
local csv,fmt,get,gt                 = _.csv,_.fmt,_.get,_.gt
local klass,lines,lt,many,map        = _.klass,_.lines,_.lt,_.many,_.map
local obj,per,push,rand,rev,rnd      = _.obj,_.per,_.push,_.rand,_.rev,_.rnd
local rogues,same,shuffle,slice,sort = _.rogues,_.same,_.shuffle,_.slice,_.sort
local values,words                   = _.values,_.words

--- learning modules
local bins,half,how

---- Klasses
local ABOUT, DATA, NOM = klass"ABOUT", klass"DATA", klass"NOM"
local RATIO, ROW,  XY  = klass"RATIO", klass"ROW",  klass"XY"

---- ---- ---- ---- Classes
-- In this code,  function arguments offer some type hints.
-- `xs` denotes a list of type `x` for
-- x in bool, str, num, int or one of the user defined types.
-- `t` denotes a list of any type. User-defined types are create by functions
-- with UPPER CASE names. Any argument with spaces before it is optional.
-- Any arguments with more than two spaces before it are local vals (so don't use thos
e).

-- **`is` recognizes column types.**
-- These column types appear in first row of our  CSV files.
local _is={
    num   = "^[A-Z]",  -- ratio cols start with uppercase
    goal  = "[!+-]$",  -- !=klass, [+,-]=maximize,minimize
    klass = "!$",       -- klass if "!"
    skip  = ":$",       -- skip if ":"
    less  = "-$"}       -- minimize if "-"

local function _col(sName,iAt)
  sName = sName or ""
  return {n    = 0,              -- how many items seen?
          at   = iAt or 0,       -- position ot column
          txt  = sName,          -- column header
          w    = sName:find(_is.less) and -1 or 1,
          ok   = true,           -- false if some update needed
          has  = {}} end         -- place to keep (some) column values.

-- **RATIO are special COLs that handle ratios.**
-- **NOM are special COLs that handle nominals.**
function RATIO:new(  sName,iAt) return _col(sName,iAt) end

function NOM:new(  sName,iAt) return _col(sName,iAt) end

-- **ROW holds one record of data.**
function ROW:new(about,t)
  return {_about=about,       -- pointer to background column info
          cells=t,            -- raw values
          cooked=nil,         -- for (e.g) discretized values
          rank=0,             -- position between 1..100
          evaled=false} end   -- true if we touched the y-values

-- **DATA holds many `ROWs`**
--  whose values are summarized in `ABOUT`.
function DATA:new() return {rows={}, about=nil} end

-- **ABOUT is a factory for making columns from column header strings.**
-- Goals and none-gaols are cached in `x` and `y` (ignorong
-- anything that is `skipped`.
function ABOUT:new(sNames)
  local about = {names=sNames,all={}, x={}, y={}, klass=nil}
  for at,name in pairs(sNames) do
    local one = (name:find(_is.num) and RATIO or NOM)(name,at)
    push(about.all, one)
    if not name:find(_is.skip) then
      push(name:find(_is.goal) and about.y or about.x, one)
      if name:find(_is.klass) then about.klass=one end end end
  return about end

-- **XY summarize data from the same rows from two columns.**
-- `num2` is optional (defaults to `num1`).
-- `y` is optional (defaults to a new NOM)
function XY:new(str,at,num1,num2,nom)
  return {txt = str,
          at  = at,
          xlo = num1,
          xhi = num2 or num1,
          y   = nom or NOM(str,at)} end

---- ---- ---- ---- Functions for Types
---- ---- ---- Create
-- Read `filename` into a DATA object. Return that object.
local function csv2data(sFilename)
  local data=DATA()
  csv(sFilename, function(t) data:add(t) end)
  return data end

-- **Copy the structure of `data`.**
-- Optionally, add rows of data (from `t`).
function DATA:clone(t)
  local data1= DATA()
  data1:add(self.about.names)
  for _,row1 in pairs(t or {}) do data1:add(row1) end
  return data1 end

---- ---- ---- Update
-- **Add a `row` to `data`.**
-- If this is top row, use `t` to initial `data.about`.
function DATA:add(t)
  if   self.about
  then push(self.rows,self.about:add(t))
  else self.about = ABOUT(t) end end

-- **Add a row of values, across all columns.**
-- This code implements _row sharing_; i.e. once a row is created,
-- it is shared across many DATAs. This means that (e.g.) distance
-- calcs are normalized across the whole space and not specific sub-spaces.
-- To disable that, change line one of this function to
-- `local row = ROW(about,x.cells and x.cells or x)`
function ABOUT:add(t)
  local row = t.cells and t or ROW(self,t) -- ensure that "x" is a row.
  for _,cols in pairs{self.x,self.y} do
    for _,col in pairs(cols) do col:add(row.cells[col.at]) end end
  return row end

-- **Add something into one `col`.**
-- For `NOM` cols, keep a count
-- of how many times we have seen `x'. For RATIO columns,
-- keep at most `the.ratios` (after which, replace old items at random).
-- `inc` is optional (it is  little hack used during
--  discretization for very
-- for fast NOM merging).
function NOM:add(x,  num)
  if x ~= "?" then
    num = num or 1
    self.n = self.n + num
    self.has[x] = num + (self.has[x] or 0) end end

function RATIO:add(x)
  if x ~= "?" then
    local pos
    self.n = self.n + 1
    if     #self.has < the.ratios        then pos = 1 + (#self.has)
    elseif rand()    < the.ratios/self.n then pos = rand(#self.has) end
    if pos then
      self.ok=false -- the `kept` list is no longer in sorted order
      self.has[pos]=x end end end

-- **Add in `x,y` values from one row into an XY.**
function XY:add(x,y)
  self.xlo = math.min(x, self.xlo)
  self.xhi = math.max(x, self.xhi)
  self.y:add(y) end

---- ---- ---- Print
-- **Print one xy**.
function XY:__tostring()
  local x,lo,hi = self.txt, self.xlo, self.xhi
  if     lo ==  hi  then return fmt("%s == %s", x, lo)
  elseif hi ==  big then return fmt("%s >  %s", x, lo)
  elseif lo == -big then return fmt("%s <= %s", x, hi)
  else                   return fmt("%s <  %s <= %s", lo,x,hi) end end

---- ---- ---- Query
-- **Return `col.has`, sorting numerics (if needed).**
function NOM:holds() return self.has end
function RATIO:holds()
  if not self.ok then table.sort(self.has) end
  self.ok=true
  return self.has end

-- **Return `num`, normalized to 0..1 for min..max.**
function RATIO:norm(num)
  local a= self:holds() -- "a" contains all our numbers,  sorted.
  return a[#a] - a[1] < 1E-9 and 0 or (num-a[1])/(a[#a]-a[1]) end

-- **Returns stats collected across a set of `col`umns**
function DATA:mid(  nPlaces,cols,    u)
  u={}; for k,col in pairs(cols or self.about.y) do
          u.n=col.n; u[col.txt]=col:mid(nPlaces) end
  return u end

function DATA:div(  nPlaces,cols,    u)
  u={}; for k,col in pairs(cols or self.about.y) do
          u.n=col.n; u[col.txt]=col:div(nPlaces) end
  return u end

--  Mode for NOM's mid
function NOM:mid(...)
  local mode,most= nil,-1
  for x,n in pairs(self.has) do if n > most then mode,most=x,n end end
  return mode end

-- Median for RATIO's mid
function RATIO:mid(  nPlaces)
  local median= per(self:holds(),.5)
  return places and rnd(median,nPlaces) or median end

-- Entropy for RATIO'd div
function NOM:div(  nPlaces)
  local out = 0
  for _,n in pairs(self.has) do
    if n>0 then out=out-n/self.n*math.log(n/self.n,2) end end
  return places and rnd(out,nPlaces) or out end

-- sd for RATIOs
function RATIO:div(  nPlaces)
  local nums=self:holds()
  local out = (per(nums,.9) - per(nums,.1))/2.58
  return places and rnd(out,nPlaces) or out end

-- **Return true if `row1``s goals are worse than `row2:`.**
function ROW:__lt(row2)
  local row1=self
  row1.evaled,row2.evaled= true,true
  local s1,s2,d,n,x,y=0,0,0,0
  local ys,e = row1._about.y,math.exp(1)
  for _,col in pairs(ys) do
    x,y= row1.cells[col.at], row2.cells[col.at]
    x,y= col:norm(x), col:norm(y)
    s1 = s1 - e^(col.w * (x-y)/#ys)
    s2 = s2 - e^(col.w * (y-x)/#ys) end
  return s2/#ys < s1/#ys end

---- ---- ---- Dist
-- Return 0..1 for distance between two rows using `cols`
-- (and `cols`` defaults to the `x` columns).
function ROW:__sub(row2)
  local row1=self
  local d,n,x,y,dist1=0,0
  local cols = cols or self._about.x
  for _,col in pairs(cols) do
    x,y = row1.cells[col.at], row2.cells[col.at]
    d   = d + col:dist(x,y)^the.p
    n   = n + 1 end
  return (d/n)^(1/the.p) end

function NOM:dist(x,y)
    return (x=="?" or y=="?") and 1 or x==y and 0 or 1 end

function RATIO:dist(x,y)
   if     x=="?" then y=self:norm(y); x=y<.5 and 1 or 0
   elseif y=="?" then x=self:norm(x); y=x<.5 and 1 or 0
   else   x,y = self:norm(x), self:norm(y) end
  return math.abs(x-y) end

-- Return all rows  sorted by their distance  to `row`.
function ROW:around(rows)
  return sort(map(rows, function(row2) return {row=row2,d = self-row2} end),--#
              lt"d") end

---- ---- ---- Clustering
-- **Divide data according to its distance to two distant rows.**
-- Use all the `best` and some sample of the `rest`.
local half={}
function half.splits(rows)
  local best,rest0 = half._splits(rows)
  print("!",cat(sort(map(rows,function(row) if row.evaled then return row.rank end end
))))
  local rest = many(rest0, #best*the.Balance)
  local both = {}
  for _,row in pairs(rest) do push(both,row).label="rest" end
  for _,row in pairs(best) do push(both,row).label="best" end
  return best,rest,both end

-- Divide the data, recursing into the best half. Keep the
-- _first_ non-best half (as _worst_). Return the
-- final best and the first worst (so the best best and the worst
-- worst).
function half._splits(rows,  rowAbove,          stop,worst)
  stop = stop or (#rows)^the.min
  if   #rows < stop
  then return rows,worst or {} -- rows is shriving best
  else local A,B,As,Bs = half._split(rows,rowAbove)
       if   B < A
       then return half._splits(As,A,stop,worst or Bs)
       else return half._splits(Bs,B,stop,worst or As) end end end

-- Do one split. To reduce the cost of this search,
-- only apply it to `some` of the rows (controlled by `the.Some`).
-- If `rowAbove` is supplied,
-- then use that for one of the two distant items (so top-level split seeks
-- two poles and lower-level poles only seeks one new pole each time).
function half._split(rows,  rowAbove)
  local As,Bs,A,B,c,far,project = {},{}
  local some= many(rows,the.Some)
  function far(row) return per(row:around(some), the.Far).row end
  function project(row)
    return {row=row, x=((row- A)^2 + c^2 - (row- B)^2)/(2*c)} end
  A= rowAbove or far(any(some))
  B= far(A)
  c= A-B
  for n,rowx in pairs(sort(map(rows, project),lt"x")) do
    push(n < #rows/2 and As or Bs, rowx.row) end
  return A,B,As,Bs,c end

---- ---- ---- Discretization
-- **Divide column values into many bins, then merge unneeded ones**
-- When reading this code, remember that NOMinals can't get rounded or merged
-- (only RATIOS).
local bins={}
function bins.find(rows,col)
  local n,xys = 0,{}
  for _,row in pairs(rows) do
    local x = row.cells[col.at]
    if x~= "?" then
      n = n+1
      local bin = col.isNom and x or bins._bin(col,x)
      local xy  = xys[bin] or XY(col.txt,col.at, x)
      add2(xy, x, row.label)
      xys[bin] = xy end end
  xys = sort(xys, lt"xlo")
  return col.isNom and xys or bins._merges(xys,n^the.min) end

-- RATIOs get rounded into  `the.bins` divisions.
function bins._bin(ratio,x,     a,b,lo,hi)
  a = ratio:holds()
  lo,hi = a[1], a[#a]
  b = (hi - lo)/the.bins
  return hi==lo and 1 or math.floor(x/b+.5)*b  end

-- While adjacent things can be merged, keep merging.
-- Then make sure the bins to cover &pm; &infin;.
function bins._merges(xys0,nMin)
  local n,xys1 = 1,{}
  while n <= #xys0 do
    local xymerged = n<#xys0 and bins._merged(xys0[n], xys0[n+1],nMin)
    xys1[#xys1+1]  = xymerged or xys0[n]
    n = n + (xymerged and 2 or 1) -- if merged, skip next bin
  end
  if   #xys1 < #xys0
  then return bins._merges(xys1,nMin)
  else xys1[1].xlo = -big
       for n=2,#xys1 do xys1[n].xlo = xys1[n-1].xhi end
       xys1[#xys1].xhi = big
       return xys1 end end

-- Merge two bins if they are too small or too complex.
-- E.g. if each bin only has "rest" values, then combine them.
-- Returns nil otherwise (which is used to signal "no merge possible").
function bins._merged(xy1,xy2,nMin)
  local i,j= xy1.y, xy2.y
  local k = NOM(i.txt, i.at)
  for x,n in pairs(i.has) do add(k,x,n) end
  for x,n in pairs(j.has) do add(k,x,n) end
  local tooSmall   = i.n < nMin or j.n < nMin
  local tooComplex = div(k) <= (i.n*div(i) + j.n*div(j))/k.n
  if tooSmall or tooComplex then
    return XY(xy1.txt,xy1.at, xy1.xlo, xy2.xhi, k) end end

---- ---- ---- Rules
-- **Find the xy range that most separates best from rest**
-- Then call yourself recursively on the rows selected by the that range.
local how={}
function how.rules(data) return how._rules1(data, data.rows) end

function how._rules1(data,rowsAll, nStop,xys)
  xys = xys or {}
  nStop = nStop or the.stop
  if #data.rows > nStop then
    local xy = how._xyBest(data)
    if xy then
      local rows1 = how._selects(xy, data.rows)
      if rows1 then
        push(xys,xy)
        print(cat(how._evals(rowsAll)),
          xyShow(xy), how._nevaled(rowsAll),#rows1)
        return how._rules1(clone(data,rows1),rowsAll, nStop,xys) end end  end
  return xys,data end

-- Return best xy across all columns and ranges.
function how._xyBest(data)
  local best,rest,both = half.splits(data.rows)
  local most,xyOut = 0
  for _,col in pairs(data.about.x) do
    local xys = bins.find(both,col)
    if #xys > 1 then
      for _,xy in pairs(xys) do
        local tmp= how._score(xy.y, "best", #best, #rest)
        if tmp > most then most,xyOut = tmp,xy end end end end
  return xyOut end

function how._nevaled(rows,     n)
  n=0;for _,row in pairs(rows) do if row.evaled then n=n+1 end end;return n end

function how._evals(rows,     n)
  return sort(map(rows,function(row) if row.evaled then return row.rank end end)) end

-- Scores are greater when a NOM contains more of the ‘sGoal‘ than otherwise.
function how._score(nom,sGoal,nBest,nRest)
  local best,rest=0,0
  for x,n in pairs(nom.has) do
    if x==sGoal then best=best+n/nBest else rest=rest+n/nRest end end
  return  (best - rest) < 1E-3 and 0 or best^2/(best + rest) end

-- Returns the subset of rows relevant to an xy (and if the subset
-- same as ‘rows‘, then return nil since they rule is silly).
function how._selects(xy,rows)
  local rowsOut={}
  for _,row in pairs(rows) do
    local x= row.cells[xy.at]
    if x=="?" or xy.xlo==xy.xhi and x==xy.xlo or xy.xlo<x and x <=xy.xhi then
      push(rowsOut,row) end end
  if #rowsOut < #rows then return rowsOut end end

-- That’s all folks
return {the=the, help=help, csv2data=csv2data,
        ABOUT=ABOUT, COL=COL, DATA=DATA, NOM=NOM,
        RATIO=RATIO, ROW=ROW, XY=XY,
        bins=bins,  half=half,  how=how}
```